1. Compiling JavaFX classes to Java classes

Ideally, we could map each JavaFX class to a corresponding Java class, and use the corresponding Java mechanisms to implement access control, inheritance, declaration of fields, declaration of methods, abstract methods, final fields, etc. However, there are a number of differences that prevent us from simply mapping FX classes to Java classes, the most significant of which is multiple inheritance. JavaFX supports multiple implementation inheritance; Java does not. However, Java does support multiple inheritance of *types* via interfaces, and we use this mechanism extensively. Further, in order to implement the dependency management required for binding and triggers, reads and writes to JavaFX attributes must be intercepted by the runtime.

1.1. Mapping FX classes

Compiliation of a JavaFX class produces two class files, an interface and a concrete class. For an FX class Foo, the interface is named Foo$Intf and the concrete class named Foo$Impl. For the following JavaFX classes:

```
class A { ... }

class B extends A { ... }
```

The compiler produces the following Java classes:

```
public interface A$Intf[1] extends FXObject { ... }

public interface B$Intf extends A$Intf, FXObject { ... }

public class A$Impl implements A$Intf { ... }

public class B$Impl implements B$Intf { ... }
```

Note that B$Impl *does not* extend A$Impl. Because Java does not support multiple implementation inheritance, we cannot (in the general case) rely on inheritance to gain access to superclass members.

All JavaFX interfaces extend the marker interface FXObject (in package com.sun.javafx.runtime), so that the compiler can distinguish between a JavaFX class and a Java class when reading a class file. This is necessary because interfaces cannot capture method modifiers such as public, private, protected, or final, so we must use an alternate means (annotations) of recording that information.

FX classes may extend Java interfaces, in which case the FX interface is made to extend the Java interface. A different approach is required to make FX classes extend Java classes.

[1] Methods or classes with $ in their name are generated by the compiler, and consumed by compiler-added code only. In particular, Java classes that want to interoperate with JavaFX classes should not access symbols with $ in their name.

1.2. Mapping FX attributes

JavaFX classes have attributes, which are data members analogous to fields in Java classes. JavaFX attributes are managed by an abstraction called Location; each JavaFX attribute is represented internally by a subtype of Location. Location provides methods to read a variable's value (recalculating if necessary), modify a variable's value (if permitted, and notifying dependencies as necessary), and to represent dependencies between variables. (Dependencies are created by binding and triggers.)

For each attribute in a JavaFX class, there are two methods in the corresponding interface, a getter and an initializer. The getter retrieves a reference to the Location representing that attribute; the initializer creates the Location representing that attribute. The initializer should only be called when the object is being constructed. Once initialized, locations are not replaced, so clients may cache the result of location getters if they like. The following table maps JavaFX types to their corresponding Location types:

Type	**Representation**
Integer	IntLocation
Number	DoubleLocation
Boolean	BooleanLocation
String	ObjectLocation<String>
Other class type T	ObjectLocation<T>
T[]	SequenceLocation<T>

Table 1. Java types corresponding to FX Locations

For this JavaFX class:

```
class A {
    attribute a : Integer;
}

class B extends A {
    attribute b : Integer;
}
```

The corresponding interface for A contains two methods:

```
public interface A$Intf extends FXObject {
    public IntLocation get$a();
    public void init$a(IntLocation a);
}
```

Similarly, the interface for B extends A$Intf and adds two methods for attribute b. When code references attribute a of A, whether from code inside of A or out, it must be converted to use methods on the corresponding Location. It must first obtain the Location by calling get$a() on the instance of A. It can then read the value by calling XxxLocation.get(), or modify the value by calling XxxLocation.set(). (Some locations are not modifiable, such as unidirectionally bound expressions, and this will cause a runtime error.)

In the concrete class, for each attribute of this class *or any superclass*, there is a private field of the appropriate Location type, a getter implementation that simply returns the Location, and an initializer implementation that sets it up. The initializer enforces that an attribute is initialized no more than once. The role of the InitHelper will be covered in section ____.

Because of the need to implement multiple inheritance, and the fact that Java cannot express multiple implementation inheritance, subclasses must “roll up” all members from superclasses into their implementation, and do not need to explicitly extend their superclass implementation. Therefore the getter and initializer methods can be made final[2].

```
public class B$Impl implements B$Intf {
    private IntLocation a;
    private intLocation b;

    public final IntLocation get$a() { return a; }
    public final IntLocation get$b() { return b; }

    public final void init$a(IntLocation location) {
        InitHelper.assertNonNull(a, "B.a");
        initHelper.add(this.a = location);
    }

    public final void init$b(IntLocation location) {
        InitHelper.assertNonNull(b, "B.b");
        initHelper.add(this.b = location);
    }
}
```

1.2.1. Static attributes

Static attributes are mapped to static fields of the implementation class. Subclasses do *not* roll up static fields from superclasses, so in order to compile a class X, the compiler must load the .class files for all the classes superclasses in order to find the static attributes. Static attributes are tagged with the @Static annotation to facilitate easier recognition. The compiler must maintain not only the list of static fields for a class, but also in what class they are declared.

If class A declares a static attribute a, then A$Impl will have a static field called a. Accesses to static attributes may be implemented as direct field accesses; if B extends A, then code in the method bodies for B will refer to a by directly accessing A$Impl.a.

Deleted: <#>Static attributes¶
Static attributes are mapped to static fields of the implementation class. Subclasses do *not* roll up static fields from superclasses, so in order to ¶

Formatted: Bullets and Numbering

1.3. Mapping FX methods

Just as attributes for all the superclasses are “rolled up” into subclasses, the same is true with method implementations – a concrete class must provide implementations for all the methods in its interface, whether those methods are defined or overridden in the current class or not.

[2] HotSpot can devirtualize and inline through monomorphic nonfinal methods, and therefore the use of final methods should in general be reserved for cases where you actually want to prohibit overriding, but less sophisticated VMs may not be able to make such optimizations. Marking the method as final provides a hint to less sophisticated VMs that the method is monomorphic and can therefore be devirtualized and potentially inlined, which can dramatically improve performance.

1.3.1. Generated methods

For each member function, the compiler generates two separate methods in the interface and implementation classes, one for use in bind contexts, and one for use not in bind contexts. For an FX method foo(), the non-bound version is called foo(), and the bound version is called foo$bound(). It is intended that Java code be able to call the unbound version but not the bound version.

For each generated method in the interface, the compiler generates two methods in the concrete class: an instance method and a static method. The static method takes an extra parameter, whose type is the generated interface type (Foo$Intf) of the declaring class.

1.3.2. Parameter type mapping

The type of the arguments and return value in the unbound implementation is obtained by simply mapping the FX types to Java types:

Type	**Representation**
Integer	int
Number	double
Boolean	boolean
String	String
Other class type T	T
T[]	Unmodifiable List<T>

Table 2. Java types corresponding to raw Java types

The type of the arguments and return value in the bound implementation is obtained by mapping the FX types to their corresponding Location type shown in Table 1. Java types corresponding to FX LocationsTable 1.

For the following classes:

```
class Base {
  function foo(a : Integer) : Integer { a+1 }
  function moo(a : Integer) : Integer { a+2 }
}

class Subclass extends Base {
  function foo(a : Integer) : Integer { a+5 }
}
```

For Base, the compiler generates the following interface and concrete class:

```
interface Base$Intf extends FXObject {
    public int foo(int a);
    public int moo(int a);

    public IntLocation foo$bound(IntLocation a);
    public IntLocation moo$bound(IntLocation a);
}

public class Base$Impl implements Base$Intf {
    protected static int foo(Base$Intf receiver, int a) {
        return a + 1;
    }
    public static int moo(Base$Intf receiver, int a) {
        return a + 2;
    }

    public static IntLocation foo$bound(final Base$Intf receiver,
                                        final IntLocation a) {
        return IntExpression.make(new IntBindingExpression() {
            public int get() {
                return a.get() + 1;
            }
        }, a);
    }

    public static IntLocation moo$bound(final Base$Intf receiver,
                                        final IntLocation a) {
        return IntExpression.make(new IntBindingExpression() {
            public int get() {
                return a.get() + 2;
            }
        }, a);
    }

    public int foo(int a) { return foo(this, a); }
    public int moo(int a) { return moo(this, a); }

    public IntLocation foo$bound(IntLocation a) {
        return foo$bound(this, a);
    }
    public IntLocation moo$bound(IntLocation a) {
        return moo$bound(this, a); }
    }
}
```

The interface specifies unbound methods foo() and moo(), and bound methods foo$bound() and moo$bound(). The concrete class implements these, and dispatches each to a static method with the same name and with a receiver variable of the interface type (Base$Intf) prepended to the front of the argument list. The body of the bound version is, in effect, the result of binding to the block expression which defines the function body.

The use of the static methods allow classes to inherit superclass method implementations without the class actually extending the superclass. The class knows, at compile time, which is the nearest superclass that actually implements the method. This is used both for dispatching to superclass methods (e.g., super.foo()) and for generating the implementation for inherited methods which are not overridden.

The receiver parameter on the static methods takes the role of the implicit "this" variable in Java classes – it is required so that the method bodies can access class attributes or call class methods. To access attribute foo, the method body would invoke the get$foo() method on the receiver.

For the derived class Subclass, the compiler generates the following interface and concrete class, which has static method bodies only for the methods that the subclass implements, and instance methods for all the methods. The instance methods dispatch to the static method of the nearest class that implements the method, which may be the subclass or may be one of the superclasses.

```
interface Subclass$Intf extends Base$Intf { }

public class Dispatch$Impl implements Dispatch$Intf {
    public static int foo(Dispatch$Intf receiver, int a) {
        return a + 5;
    }

    public static IntLocation foo$bound(
            final Subclass$Intf receiver,
            final IntLocation a) {
        return IntExpression.make(new IntBindingExpression() {
            public int get() {
                return a.get()+ 5;
            }
        }, a);
    }

    public int foo(int a) { return foo(this, a); }
    public int moo(int a) { return Base$Impl.moo(this, a); }


    public IntLocation foo$bound(IntLocation a) {
        return foo$bound(this, a);
    }
    public IntLocation moo$bound(IntLocation a) {
        return Base$Impl.moo$bound(this, a);
    }
}
```

1.3.3. Static functions

Like static fields, static functions can be implemented as static methods on the implementation class in which they are declared, tagged with the @Static attribute. When compiling a class, the compiler needs to process the .class files for all the classes superclasses to find the static functions. The compiler should generate both a bound and unbound implementation for static functions.

Formatted: Bullets and Numbering

Formatted: Normal

1.3.4. Abstract classes

Formatted: Bullets and Numbering

For abstract classes, the compiler generates the interface just as for instantiable classes. The concrete class should be generated as abstract, and need only generate the static methods corresponding to instance methods; it need not generate any instance methods as the class will never be instantiated. Abstract classes, therefore, exist only as repositories for inherited implementations that can be dispatched to by instance methods of subclasses.

1.4. Object initialization

Unlike Java classes, JavaFX classes do not have constructors (other than the default no-arg constructor.) Java constructors are responsible for initializing the object in a known consistent state, and not allowing clients to use the object unless it is able to do so; JavaFX classes, by contrast, are initialized by allowing clients to specify an arbitrary subset of the classes attributes through object literals. (This often requires classes to expose more of their attributes to clients.)

Object literals take the form:

```
var v = Foo { a: 3, b : 4 }
```

Compiler-generated client code is responsible for managing the lifecycle of object allocation and initialization. The steps are:

- Instantiate an empty object with the no-arg constructor
- Call the initializer method to set any attributes that have been specified in the object literal
- Call the initialize$() method on the newly initialized object to complete initialization

For the above object literal, the compiler would generate the following client code:

```
Foo$Impl tmp = new Foo$Impl();
tmp.init$a(IntVar.make(3));
tmp.init$b(IntVar.make(4));
tmp.initialize$();
ObjectLocation<Foo$Impl> v = ObjectLocation<Foo$Impl>.make(tmp);
```

1.4.1. Bound attributes

Object literals may specify bindings as well as concrete values:

```
var n = 1;
var v = Foo { a: 3, b : bind n+1 }
```

In this case, the only difference is that instead of creating variable locations (XxxVar), the generated code creates expression locations (XxxExpression), just as if the binding was being used in a variable initializer:

```
final IntLocation n = IntVar.make(1);
Foo$Impl tmp = new Foo$Impl();
tmp.init$a(IntVar.make(3));
tmp.init$b(IntExpression.make(new IntBindingExpression() {
               public int get() { return n.get() + 1; }
           }, n));
tmp.initialize$();
ObjectLocation<Foo$Impl> v = ObjectLocation<Foo$Impl>.make(tmp);
```

Comment [BG1]: These locations may not yet be available; the addDependencies() call should come between the call to setDefaults$() and the initial firing of change triggers.

1.4.2. Default attribute values

Attribute declarations may include an optional initializer expression, which provides the *default* value for the attribute only in the event that the attribute is not initialized by the object literal.

For each concrete class Foo$Impl, the compiler must generate a static setDefaults$(Foo$Intf) method that initializes attributes with their default value. This is called from the initialize$() method, after the object literal values have been set, so this method can inspect the attribute location to see if it is null (not already set by the object literal) and conditionally initialize the attribute. For the following class:

```
class Foo {
    attribute a : Integer = 3;
}
```

The compiler would generate a setDefaults$() method to conditionally initialize a:

```
class Foo$Impl implements Foo$Intf {
    ...
    public static void setDefaults$(final Foo$Intf receiver) {
        if (receiver.get$a() == null)
            receiver.init$a(IntVar.make(3));
    }
    ...
}
```

1.4.3. Class initializers

Constructors in Java classes have a chance to inspect, validate, and reject the constructor arguments, and to initialize private state based on the provided arguments. Constructors in JavaFX classes run before the object literal values are applied to the object, so these tasks cannot be performed from constructors. JavaFX classes have initializer blocks, which are blocks of code that are executed during initialization, *after* object literal values and default values have been applied.

```
class Foo {
    attribute size : Integer = 3;
    private attribute map : HashMap;

    init {
        map = new HashMap(size);
        System.out.println("Initialized Foo!");
    }
}
```

For each class, the compiler generates a static userInit$() method, which contains the code corresponding to the init block. If there is no init block, the compiler must generate an empty userInit$() method.

```
class Foo$Impl implements Foo$Intf {
    ...
    public static void userInit$(final Foo$Intf receiver) {
        receiver.init$map(ObjectVar<Map>.make(
            new HashMap(receiver.get$a().get()));
        System.out.println("Initialized Foo!");
    }
    ...
}
```

1.4.4. The initialize$() method and the initialization helper

Every class must have an initialize$() method, which performs the following steps:

- Sets default values for attributes not initialized by the object literal
- Run the class initializers
- Fire the change triggers for any attribute set via either the object literal or the default value

The compiler should also generate an initialization helper object as part of the class, and initialize it, and call its initialize() method at the end of initialize$(). This will cause the change triggers to be fired, firing for object-literal-provided attributes before default-provided values. (This is important in cases where attributes have change triggers that overwrite other attributes.) The initialize$() method should always look as follows:

```
class Foo$Impl implements Foo$Intf {
    private static final int NUM$FIELDS = 2;
    private InitHelper initHelper = new InitHelper(NUM$FIELDS);
    ...
    public void initialize$() {
        setDefaults$(this);
        userInit$(this);
        initHelper.initialize();
        initHelper = null;
    }
}
```

The initialization of InitHelper requires knowing how many attributes there are in the class; the compiler should generate the static final field NUM$FIELDS and a getter called getNumFields$() to indicate this. For classes with superclasses, this should be generated by adding the number of fields declared in this class to the NUM$FIELDS value from each superclass:

```
    private static final int NUM$FIELDS = 2
                         + Super$Impl.getNumFields$();
```

1.4.5. Object initialization and inheritance

In the presence of inheritance, the userInit$() and setDefaults$ methods must call the corresponding method for all superclasses. For the following classes:

```
class A {
    attribute a : Integer = 1;
    init { System.out.println("A"); }
}
class B {
    attribute b : Integer = 1;
    init { System.out.println("B"); }
}
class C extends A, B {
    attribute c : Integer = 1;
    init { System.out.println("C"); }
}
```

The compiler would generate the userInit$() and setDefaults$() method for C$Impl as follows:

```
class C$Impl implements C$Intf {
    ...
    public static void userInit$(final C$Intf receiver) {
        A.userInit$(receiver);
        B.userInit$(receiver);
        System.out.println("C");
    }

    public static void setDefaults$(final C$Intf receiver) {
        if (receiver.get$c() == null)
            receiver.init$c(IntVar.make(1));
        A.setDefaults$(receiver);
        B.setDefaults$(receiver);
    }
    ...
}
```

The rationale for separating the implementation of setting defaults and class initialization is that we want all the defaults to be set before any of the class initializations to execute; if there was one initialize method that did both, superclass class initializers could run before the subclass had a chance to set default values.

1.4.6. Change listeners

Class attributes may have change listeners, specified by the “on replace”, “on insert”, or “on delete” clauses on the attribute declaration. For each change listener declared, the compiler must generate a change listener and add it to the attribute’s Location. This can be done in the setDefaults$ method, after all default values have been applied.

```
class Foo {
    attribute size : Integer = 3
        on replace { System.out.println("a={a}"); };
}
```

The compiler would generate the setDefaults$ method as follows:

```
protected static void setDefaults$(final Foo$Intf receiver) {
        if (receiver.get$a() == null)
            receiver.init$a(IntVar.make(3));

        receiver.get$a().addChangeListener(new ChangeListener() {
            public boolean onChange(Location location) {
                System.out.println("a=" + receiver.get$a().get());
                return true;
            }
        });
    }
```

1.5. Method invocation and return values

The mechanics of parameter passing varies depending on whether or not the function application appears in a bind context. There are several parameter-passing protocols that may be used both for parameters and return values: by value, by unmodifiable location, or by reference.

All JavaFX local variables and object attributes are represented by a Location. Location has methods for retrieving the current value, modifying the current value, and registering dependencies on the value. Some Locations are modifiable (e.g., those that correspond to non-final variables), and some are unmodifiable (e.g., those that correspond to computed values or to final variables.)

1.5.1. Pass-by-value

All values in JavaFX – primitives, object references, sequences – are immutable. In pass-by-value, the argument expression or function block is evaluated, and the resulting value passed using the types in Table 2.

1.5.2. Pass-by-reference

In pass-by-reference, the Location representing the actual argument (if the actual is an lvalue) is passed to the method implementation. This allows the method body to modify the value, and to register dependencies or change triggers on the variable being passed.

1.5.3. Pass-by-unmodifiable-location

In pass-by-unmodifiable-location, a wrapped Location is passed to the method body. The wrapper allows the method body to register dependencies or change triggers on the underlying Location, but not to modify its value. Pass-by-unmodifiable-location is used when the function is invoked in a bind context, or explicit binding is requested by the caller.

1.5.4. Unbound function invocation

For functions that are invoked not in a bind context:

```
var v = object.foo(a, b, c+d);
```

The unbound version of the function is invoked, and parameters are passed by value. The actual parameters are evaluated by the caller and the values are passed to the unbound implementation of foo(). The return value is returned by value.

1.5.5. Bound function invocation

For functions that are invoked in a bind context:

```
var v = bind object.foo(a, b, c+d);
```

The bound version of the function is invoked, and parameters are passed by unmodifiable location. If one of the actual parameters is a constant expression or an expression derived from other variables, the client creates a temporary unmodifiable location to describe it, and passes that. The return value from the bound version of the function is passed by unmodifiable location as well.

1.5.6. Bound parameters

There may be cases where the return value of a function is not bound (the function is not evaluated in a bind context) but the client wishes to pass one or more of the parameters by location. In this case, the client can specify the bind keyword on the actual parameter:

```
var v = object.foo(bind a, b, c+d);
```

In this case, the bound version of the method is called, using temporary locations for the unbound parameters, and the return value is unboxed from the returned location, and the returned location is discarded.

1.5.7. Return value binding

Function bodies are defined by block expressions. The unbound version of the function simply evaluates the block expression in the function's closure. The bound version of the function evaluates

```
var t = bind { function-block-expression }
```

in the function's closure. The result of the bind expression is a Location, which is returned to the caller.

1.6. Multiple inheritance and duplicate names

If a class extends multiple classes, it is possible that some members (attributes or methods) will be present in more than one superclass. Duplicate members are allowable so long as their signatures are compatible.[3] When there is a duplicated member, the subclass has only one member corresponding to the duplicated member. When there are duplicate but compatible attributes, the compiler should emit a warning; when the superclass members are incompatible, the compiler should emit an error.

[3] A detailed definition of compatibility is required.

1.6.1. Attribute conflicts

Comment [BG2]: This is going to have to be fixed. One way to fix this is to include the superclass name in the getter/initializer method (i.e., get$Foo$foo() instead of get$Foo()).

Warnings related to attribute conflicts are serious, because they indicate that multiple implementations unknowingly share a variable that they each think they own exclusively. This is almost always a problem, unless both superclasses themselves inherit from a common superclass (the so-called "diamond of derivation").

This problem is made worse by the fact that private attributes are not truly private; they still appear in interfaces, and therefore it is quite likely that there will be inadvertent name collisions in multiple inheritance. Further, because these names are part of interfaces, it is not possible to "rename" them in subclasses. Classes designed for multiple inheritance (such as "mixin" classes) should choose names that are unlikely to conflict.

1.6.2. Initializer conflicts

One possible conflict is when multiple superclasses provide a default value for a duplicated attribute. In this case, the class that appears first in the list of superclasses wins, because the setDefaults$() methods are called on superclasses in the order they are defined in the source code.

1.6.3. Method conflicts

Another possible conflict is when multiple superclasses provide an implementation of a method. In this case, the subclass can disambiguate which method it wants to call by explicitly specifying A.super.foo() or B.super.foo() (where A and B are the superclass names.) Just as for initializer conflicts, method conflicts are resolved in favor of the class that appears first in the list of superclasses. (The subclass can override the method and dispatch to the losing superclass method if this default isn't the desired choice.)

1.6.4. Diamond-shaped inheritance graphs

It is possible, with multiple inheritance, to have diamond-shaped inheritance graphs:

```
class A { ... }

class B extends A { ... }

class C extends A { ... }

Class D extends B, C { ... }
```

In this case, all the members of A are duplicated through B and C. In the simple case, where B and C do not override any of the members of A (or attempt to provide default initial values for A's attributes), then the duplicate-resolution approach from the previous section yields the same result as if A were a virtual base class in C++; D inherits one copy of each of the members of A.

In the case where B and C override some of the members of A, then conflicts are resolved according to the same mechanism as for duplicated members.

1.7. Access control

Methods on interfaces are implicitly public. This means we cannot piggyback on Java's access control mechanisms. Instead, all members are public in the generated Java classes. Access control is enforced instead by the JavaFX compiler.

For each visibility or other modifier that can appear on class members (e.g., public, private, protected, abstract, final), there is a corresponding annotation in com.sun.javafx.runtime. For each private attribute or method, the JavaFX compiler applies the @Private attribute to the interface methods related to that member. When the compiler loads a class that extends FXObject, it should look for these annotations and build visibility and other modifier information into its symbol table.

1.8. Inner classes